Introduction to Embedded Systems

CS/ECE 6780/5780

Al Davis

Today's topics:

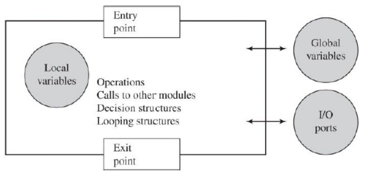
- logistics – minor
- more software development issues

Software Update Problem

- **Lab machines work**
  - let us know if they don't
- **Personal machines have update issues**
  - Torrey & William have been trying to find a fix
    - » so far the solution has been elusive
- **Labs**
  - next lab will be assembly oriented
  - hopefully posted by tomorrow

Modular SW Development

- **Modular programming**
  - breaks SW into distinct and independent modules
  - provides:
    - » functional abstraction to support reuse
    - » complexity abstraction
      - divide and conquer approach
    - » portability



Global Variables

- **Global information**
  - shared by more than one module
  - use them to pass data between ***main()*** and ***interrupts***
  - data is permanent and not deallocated
- **Can use absolute addressing to access globals**
- **I/O ports and registers are considered global values**
  - it would be silly to view them as locally scoped

Local Variables

- **Temporary information used by only one module**
  - **typically allocated, used, and deallocated**
    - » **hence information is not permanent**
- **Stored on stack or registers**
  - **dynamic allocation/release allows memory reuse**
  - **limited scope provides data protection**
  - **since interrupt saves registers and uses it's own stack**
    - » **code may still be re-entrant**
  - **code is relocatable**
  - **number of local variables**
    - » **only limited by stack size**

Two Local 16-bit Variables: Take 1

```
;unsigned short calc(void){ unsigned short sum,n;
;  sum = 0;
;  for(n=100;n>0;n--){
;    sum=sum+n;
;  }
;  return sum;
;}
; *****binding phase***********
sum  set  0  16-bit number
n    set  2  16-bit number
; *******allocation phase *****
calc pshx   ;save old Reg X
     pshx   ;allocate n
     pshx   ;allocate sum
     tsx    ;stack frame pointer
```

*X ➔ sum and *X+2 ➔ n (stack grows down)
n and sum are now X relative offsets

Take 1 continued

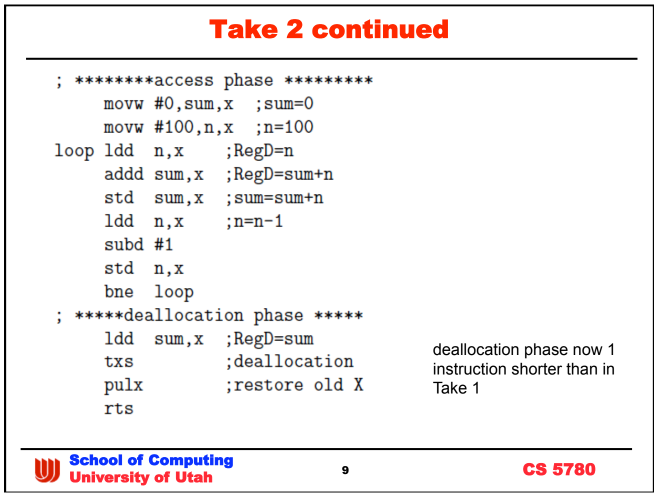
```
; ********access phase ********
     ldd  #0
     std  sum,x  ;sum=0
     ldd  #100
     std  n,x    ;n=100
loop ldd  n,x    ;RegD=n
     addd sum,x  ;RegD=sum+n
     std  sum,x  ;sum=sum+n
     ldd  n,x    ;n=n-1
     subd #1
     std  n,x
     bne  loop
; ******deallocation phase ***
     ldd  sum,x  ;RegD=sum
     pulx        ;deallocate sum
     pulx        ;deallocate n
     pulx        ;restore old X
     rts
```

Two Local 16-bit Variables: Take 2

- **6812 allows negative offset addressing**

```
; *****binding phase************
sum  set  -4  16-bit number
n    set  -2  16-bit number
; *******allocation phase ******
calc pshx       ;save old Reg X
     tsx        ;stack frame pointer
     leas -4,sp ;allocate n,sum
```

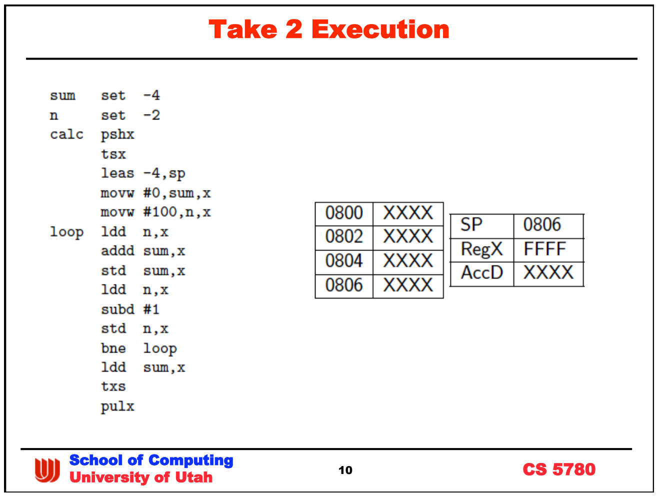
X now points inside the stack frame
leas: load effective address ➔ SP



Take 2 continued

```
; ********access phase *********
     movw #0,sum,x   ;sum=0
     movw #100,n,x   ;n=100
loop ldd  n,x     ;RegD=n
     addd sum,x   ;RegD=sum+n
     std  sum,x   ;sum=sum+n
     ldd  n,x     ;n=n-1
     subd #1
     std  n,x
     bne  loop
; *****deallocation phase *****
     ldd  sum,x   ;RegD=sum
     txs          ;deallocation
     pulx         ;restore old X
     rts
```

deallocation phase now 1 instruction shorter than in Take 1

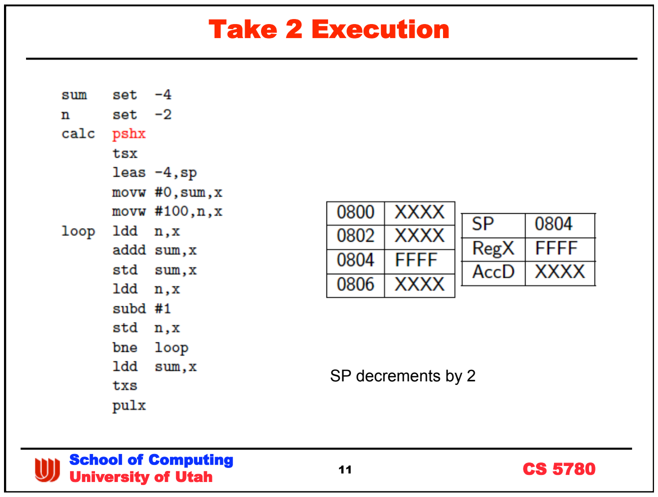


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	XXXX
0802	XXXX
0804	XXXX
0806	XXXX

SP	0806
RegX	FFFF
AccD	XXXX



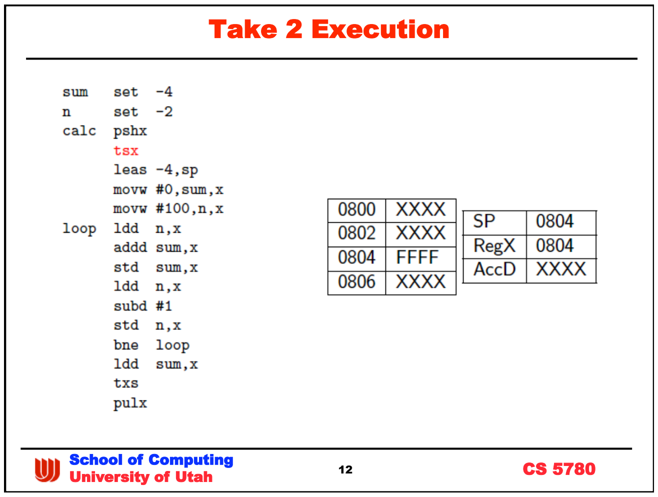
Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	XXXX
0802	XXXX
0804	FFFF
0806	XXXX

SP	0804
RegX	FFFF
AccD	XXXX

SP decrements by 2

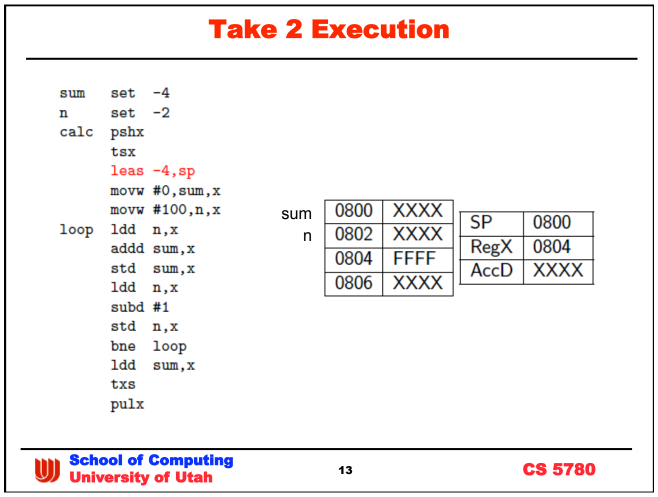


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	XXXX
0802	XXXX
0804	FFFF
0806	XXXX

SP	0804
RegX	0804
AccD	XXXX

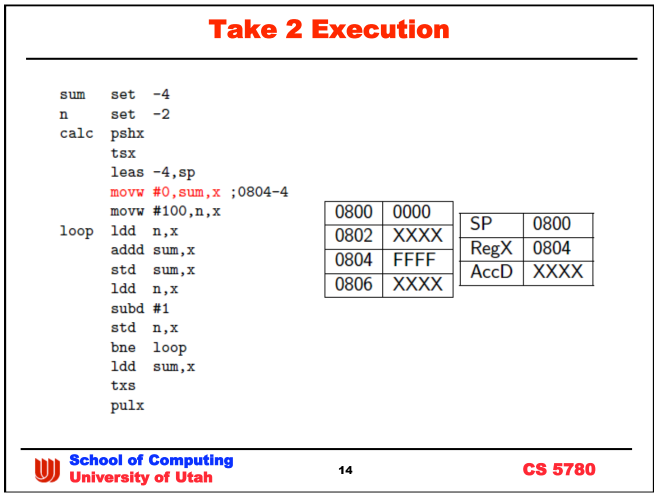


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

sum	0800	XXXX
n	0802	XXXX
	0804	FFFF
	0806	XXXX

SP	0800
RegX	0804
AccD	XXXX

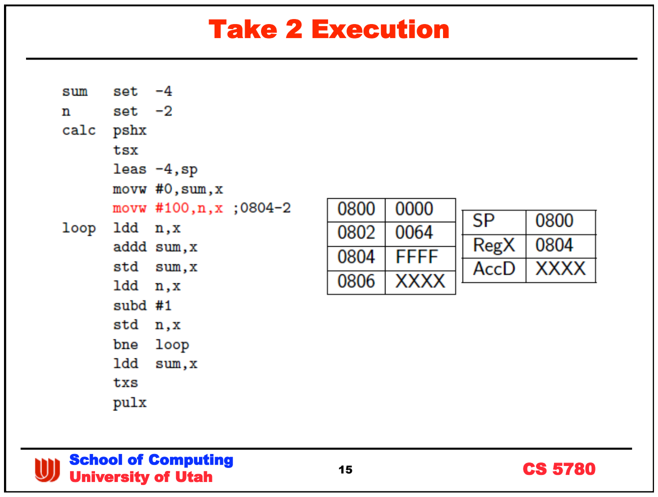


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x ;0804-4
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0000
0802	XXXX
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	XXXX

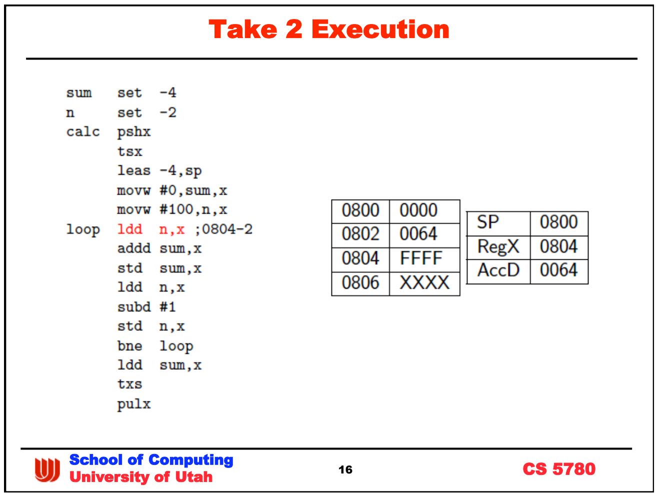


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x ;0804-2
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0000
0802	0064
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	XXXX

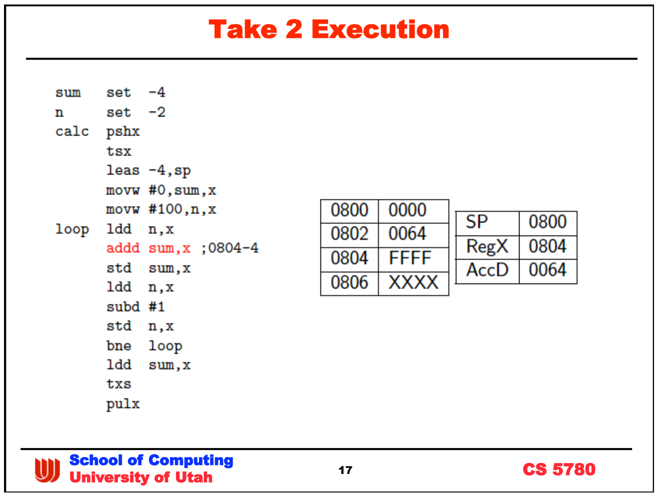


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x ;0804-2
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0000
0802	0064
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	0064

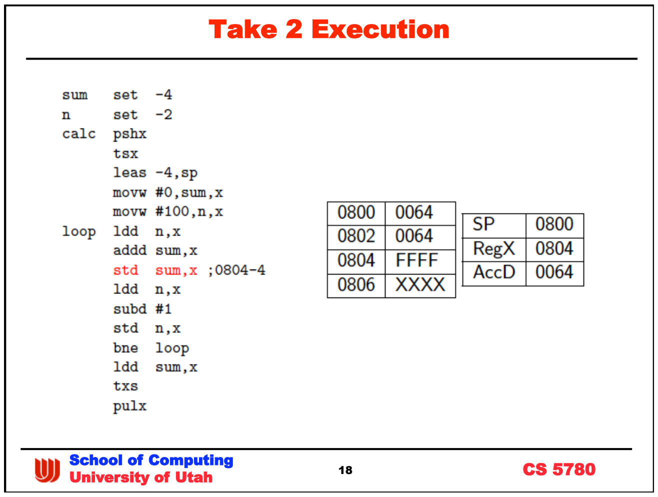


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x  ;0804-4
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0000
0802	0064
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	0064

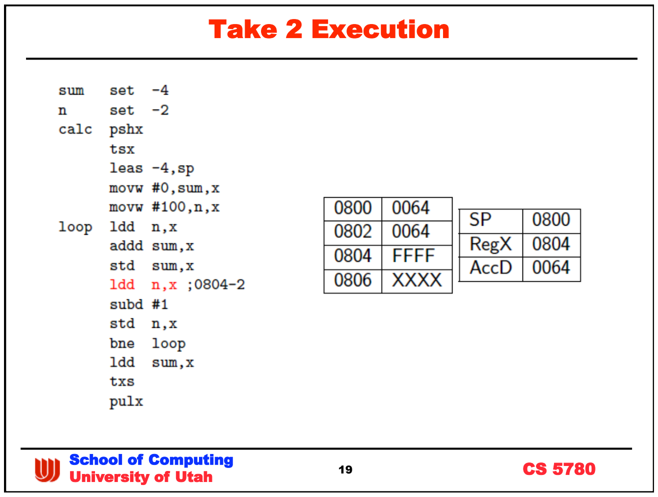


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x  ;0804-4
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0064
0802	0064
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	0064

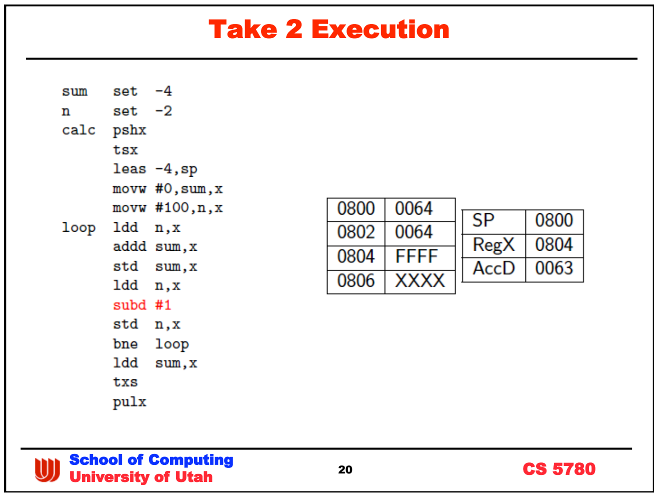


Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x  ;0804-2
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0064
0802	0064
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	0064



Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0064
0802	0064
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	0063

Take 2 Execution

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x ;0804-2
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	0064
0802	0063
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	0063

End of loop

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	13BA
0802	0000
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	0000

Put Sum in D register

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x ;0804-4
      txs
      pulx
```

0800	13BA
0802	0000
0804	FFFF
0806	XXXX

SP	0800
RegX	0804
AccD	13BA

Restore Stack Pointer & Dellocate

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	13BA
0802	0000
0804	FFFF
0806	XXXX

SP	0804
RegX	0804
AccD	13BA

Restore X Register & Voila done

```
sum   set  -4
n     set  -2
calc  pshx
      tsx
      leas -4,sp
      movw #0,sum,x
      movw #100,n,x
loop  ldd  n,x
      addd sum,x
      std  sum,x
      ldd  n,x
      subd #1
      std  n,x
      bne  loop
      ldd  sum,x
      txs
      pulx
```

0800	13BA
0802	0000
0804	FFFF
0806	XXXX

SP	0806
RegX	FFFF
AccD	13BA

Returning Multiple Parameters: Registers

```
module: ldaa #1
        ldab #2
        ldx  #3
        ldy  #4
        rts      ;returns 4 parameters in 4 registers
********calling sequence******
        jsr module
* Reg A,B,X,Y have four results
```

Returning Multiple Values: stack

```
data1   equ 2
data2   equ 3
module: movb #1,data1,sp ;1st parameter onto stack
        movb #2,data2,sp ;2nd parameter onto stack
        rts
*******calling sequence******
        leas -2,sp ;allocate space for results
        jsr module
        pula  ;1st parameter from stack
        staa first
        pula  ;2nd parameter from stack
        staa second
```

More Issues

- **All assembly exit points**
  - **must balance the stack**
    - **call and return sequences are mirrored**
- **Performing unnecessary I/O in a subroutine**
  - **limits reuse**
- **I/O devices must be considered global**
  - **restrict the number of modules that access them**
- **Information hiding**
  - **expose only the necessary information at the interfaces**
    - **promotes understanding and reduces conceptual complexity**
    - **e.g. hide inner workings of the black box from the user**

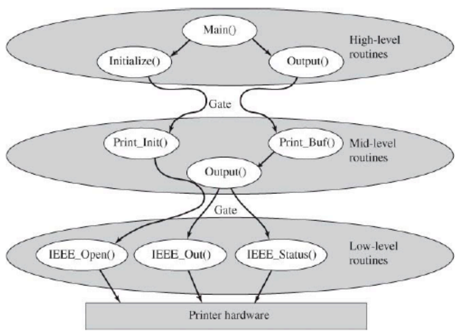
Module Decomposition

- **Coupling**
  - influence a module's behavior has on another module
- **Task decomposition goals**
  - make the SW organization easier to understand
  - increase the number of modules
    - » this may increase code footprint and/or increase run time
      - due to extra subroutine linkages
    - » but start properly and then optimize if you have to
    - » minimize coupling as much as possible
- **Develop, connect and test modules in a hierarchy**
  - top-down – "write no software until every detail is specified"
  - bottom-up – "one brick at a time"
- **Initial design is best done top-down**
- **Implementation is best done bottom-up**
  - namely you have something to test

Layered Software Systems

- **Note**
  - SW continually changes as better HW or algorithms become available
- **Layered SW facilitates these changes**
  - top layer is the main program
  - lowest layer is the HW abstraction layer
    - » modules that access the I/O HW
- **Hierarchy should be strict**
  - each layer can only call lower layers
    - » ideal is to only call the next lower layer
  - gate or API
    - » defines the interface at the next lower layer
  - if this happens
    - » each layer can be replaced without affecting other layers
    - » possible downside: code bloat
      - optimize last policy is a good one
      - easier to optimize correctly based on measurements of working code

Layered Parallel Port Example



Layered SW Rules

- **Modules may make calls to modules in the same layer**
- **Modules may call lower layer only using gate**
- **Module has no access to any function or variable in another layer**
  - except via gate
- **Modules can't call upper layers**
- **Ideal yet optional**
  - module calls only next layer down
  - all I/O access is at the lowest layer
  - user interface is at the highest level

Device Driver Concepts

- Purpose
  - SW interface to physical I/O devices
    - » interface API for upper layers
    - » low-level routines to configure and perform actual I/O
- Separation of policy and mechanism is important
  - e.g. interface may include routines to open, read, and write files
    - » but it shouldn't care about what device the files reside on
- HAL
  - provide a good hardware abstraction layer

Low-Level Device Drivers

- Normally found in BIOS ROM
  - basic I/O system
- Good low-level drivers allow:
  - new hardware to be installed
  - new algorithms to be implemented
    - » synchronization w/ completion flags/interrupts
    - » error detection and recovery methods
  - higher-level features built on top of low-level
    - » OS features like blocking semaphores
    - » driver features like automatic compression/decompression

Encapsulated Objects in ANSI C

- Choose names to reflect the module in which they are defined
  - Example

In C: `LCD_Clear()`
In C++: `LCD.clear()`

Only put public function declarations in header files.

Example (`Timer.H`):

```
void Timer_Init(void);
void Timer_Wait10ms(unsigned short delay);
```

Since the function `wait(unsigned short cycles)` is not in the header file, it is a private function.

Reentrancy

- Reentrant if
  - it can be conurrently executed by 2 or more threads
  - or by main and one or more interrupts
- Rules for reentrant functions
  - must not call a non-reentrant function
  - must not touch global variables w/o proper locking

Coding Guidelines

- Guidelines that cannot be checked by a smart compiler are less effective
- Too many guidelines are worthless
  - too hard to remember or enforce
- Following is a 10 rule list
  - by Gerard Holzman
    - » leads NASA/JPL's Lab for Reliable Software
      - needless to say it's hard to debug things in outer space
  - note
    - » these are good things to know
    - » even though some of the implied tools aren't available to you at the moment

Rule 1

Rule: Restrict all code to very simple control flow constructs – do not use *goto* statements, *setjmp* or *longjmp* constructs, and direct or indirect *recursion*.

Simple control translates into easier code verification and often improved clarity.

Without recursion the function call graph is acyclic which directly aids in proving boundedness of the code.

This rule doesn't require a single return point for a function although this often simplifies control flow.

Rule 2

Rule: All loops must have a fixed upper-bound. It must be trivially possible for a checking tool to *prove* statically that a preset upper-bound on the number of iterations of a loop cannot be exceeded. If the loop-bound cannot be proven statically, the rule is considered violated.

The absence of recursion and presence of loop bounds prevents runaway code.

Functions intended to be nonterminating must be proved to *not* terminate.

Some functions don't have an obvious upper bound (i.e. traversing a linked list), so an artificial bound should be set and checked via an assert.

Rule 3

Rule: Do not use dynamic memory allocation after initialization.

Memory allocation code is unpredictable from a time standpoint and therefore impractical for time critical code.

Many errors are introduced by improper dynamic memory allocation.

Without dynamic memory allocation the stack is used for dynamic structures and without recursion bounds can be proved on stack size.

Rule 4

Rule: No function should be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than 60 lines of code per function.

Long functions are like run-on sentences: they need to be rewritten.

Rule 5

Rule: The *assertion density* should average to a minimum of two assertions per function. Assertions are used to check for anomalous conditions that should never happen in real-life executions. Assertions must always be side-effect free and should be defined as Boolean tests. When an assertion fails, an explicit recovery action must be taken.

Use of assertions is recommended as part of a strong defensive coding strategy.

Assertions can be used to check pre- and post-conditions of functions, parameter values, return values, and loop invariants.

Assertions can be disabled in performance critical code because they are side-effect free.

Rule 6

Rule: Data objects must be declared at the smallest possible level of scope.

Variable will not be modified in unexpected places if they are not in scope.

It can be easier to debug a problem if the scope of the variable is smaller.

Rule 7

Rule: The return value of non-void functions must be checked by each calling function, and the validity of parameters must be checked in each function.

If the response to the error would be no different to the response to the success then there is no point in checking the value.

Useless checks can be indicated by casting the return value to (void).

Rule 8

Rule: The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, variable argument lists, and recursive macro calls are not allowed. All macros must expand into complete syntactic units. The use of conditional compilation directives is often also dubious but cannot always be avoided. Each use of a conditional compilation directive should be flagged by a tool-based checker and justified in the code.

Conditional compilation directives can result in an exponentially growing number of code versions.

Rule 9

Rule: The use of pointers should be restricted. Specifically, no more than one level of dereferencing is allowed. Pointer dereference operations may not be hidden in macro definitions or inside *typedef* declarations. Function pointers are not permitted.

Pointers are easily misused even by experienced programmers.

Function pointers can severely limit the utility of static code checkers.

Rule 10

Rule: All code must be compiled, from the first day of development, with *all* compiler warnings enabled at the compiler's most pedantic setting. All code must compile with these settings without any warnings. All code must be checked daily with at least one, but preferably more than one, state-of-the-art static code analyzer and should pass the analyses with zero warnings.

This rule should be followed even in the case when the warning is invalid.

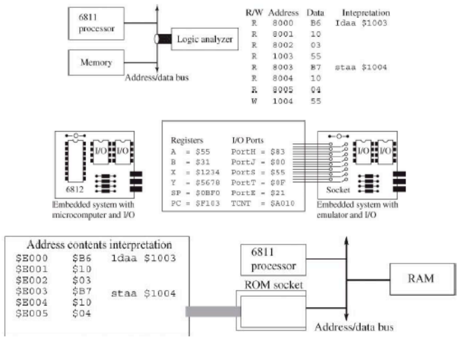
Code that confuses the compiler or checker enough to result in an invalid warning should be rewritten for clarity.

Static checkers should be required for any serious coding project.

Debugging Theory

- **Process of testing, stabilizing, localizing, and correcting errors**
- **Research in program monitoring & debugging has not kept up**
  - **gdb has been around for 30+ years**
  - **so has printf**
  - **alas the Symbolics 3600 system has been left behind**
    - **» it shouldn't have**
- **ES debugging is even more complicated**
  - **by concurrency and real-time requirements**
  - **printf is a problem because they are slow**

HW Debugging



Debugging w/ SW

- **Debugging instrument**
  - » **code added to a program to improve visibility of internals**
  - » **extra visibility aids debugging**
  - » **printf is the common example**
- **Printf instrument policy (use one or more)**
  - **place printf statements in a unique column**
  - **define instruments with a specific naming pattern**
  - **define all instruments to test a run-time global flag**
  - **use conditional compilation (assembly) to turn on/off**

Functional/Static Debugging

- **Functional → check that the right computation is done**
  - **Inputs supplied**
  - **run system**
  - **check outputs**
- **Several methods**
  - **single step**
  - **tracing**
  - **breakpoints w/o filtering**
  - **conditional breakpoints**
  - **instrumentation: printf's to a trace file**
    - » **with or without filtering**
      - **you only want values within a specific range**
  - **monitor with a fast display**

Performance Debugging

- **Verification of timing behavior**
  - **run system and check dynamic I/O behavior**
    - » **count bus cycles using the assembly listing**
    - » **instrumentation – measure with a counter**

```
unsigned short before,elasped;
void main(void){
   ss=100;
   before=TCNT;
   tt=sqrt(ss);
   elasped=TCNT-before;
}
```

Instrumentation via Output Port

```
Set bset PORTB,#$40
    rts
Clr bclr PORTB,#$40
    rts

loop jsr  Set
     jsr  Calculate   ; function under test
     jsr  Clr
     bra  loop
```

How would you improve on this?

Concluding Remarks

- As Arlo says
  - "you can't always do what you're supposed to do"
- But
  - keeping these coding tips in mind will make you a better programmer
  - and reduce hair pulling panic attacks
  - in later professional life
    - » these types of things will be mandated by your company
      - albeit in a slightly different form
- So
  - might as well develop good habits early
    - » some of you already have